Building an OpenStep Application

by Michael Rutman, independent consultant

Is it really as easy to program in OpenStep

OpenStep and NEXTSTEP

In 1985, Steve Jobs left Apple and formed NeXT. He found the latest technologies and brought together a team of developers to turn the newest theories into realities. To verify the new technologies, during development Steve would often stop development and have the entire team use the system they were creating. Several existing apps were created during these day or week long kitchens. More importantly, each developer knew how the framework would be used. If developers had a hard time using an object during a kitchen, they knew they had to rework that object.

The system they created was called NEXTSTEP. NEXTSTEP has several layers, and each layer was state of the art in 1985. In the 12 years since Steve picked these technologies, the state of the art may have moved, but not advanced.

The underlying OS is a Mach mini-kernel running a unix emulator. For practical purposes, Mach is a flavor of BSD unix. However, the Mach mini-kernel offers programmers a rich set of functionality beyond what unix provides. Most of the functionality of Mach is wrapped into the NEXTSTEP framework, so programmers get the power and flexibility of Mach with the ease of use of NEXTSTEP.

Sitting on top of Mach is Display Postscript. Postscript, the language of printers, is a nice graphics language. NeXT and Adobe optimized Postscript for displaying on the screen and produced a speedy and powerful display system. NeXT's framework hides most of the Postscript, but if a programmer wants to get into the guts, there are hooks, called pswraps, to work at the Postscript level.

NEXTSTEP needed a language, and deciding on a language was difficult. In 1985, there were a lot of theoretical languages that claimed to be Object-Oriented, but few were in actual use. C++ was not shipping, but Objective-C was. Furthermore, Objective-C was a much easier language to learn and use. Unfortunately, the rest of the industry went with C++. NeXT responded by adding C++ to Objective-C. Today, under OPENSTEP, programmers can use Objective-C, Java and C++. However, the framework is still based on Objective-C, so learning Objective-C will be important.

Using Objective-C, NeXT created the AppKit. The AppKit is a framework for creating applications. There are many frameworks for making applications today, but the original AppKit is the easiest to use. In addition to the AppKit, NeXT expanded with the Music Kit, Foundation Kit, and EOF (Database Kit). With these kits, a developer can create full featured applications very quickly.

The last layer of NEXTSTEP is the tools. NeXT has implemented a very nice suite of developer tools. NeXT has separate applications for each part of development, but the applications communicate with each other providing an integrated environment. Source files are edited in Edit.app, Projects are maintained and compiled in ProjectBuilder.app, Interfaces are created and edited in InterfaceBuilder.app, and debugging is done with GDB from Edit.app. In addition, there are command line alternatives for any of these GUI applications. Each of these applications work with the command line programs, so going between the two is painless. Personally, I spend most of my time in the GUI applications, but I know other programmers who use emacs exclusively.

Now with NEXTSTEP defined, what is OPENSTEP? OPENSTEP is the same as NEXTSTEP, but does not include the development tools or the OS. OPENSTEP will run under any OS, and any development tools can be used to create OPENSTEP applications. At least that is the theory - we will see what Apple decides for Rhapsody. At the time of writing this article, it looks like Apple's Rhapsody will be close to NEXTSTEP.

OPENSTEP comes with tutorials to explain exactly how to use the development tools, so I will gloss over the details where they are straight-forward. I will create a networked lunch application using NEXTSTEP 3.3, which will be similar to OPENSTEP, but not quite. Each person on the network will be able to say where they want to eat, and everyone else running the application will see their choices. In general, development under NEXTSTEP is identical to development under OPENSTEP.

Creating an Application

All OPENSTEP applications start with ProjectBuilder.app. After launching ProjectBuilder.app, we create a new application called "lunch." ProjectBuilder also lets you create bundles, palettes, tools, libraries, and more. Starting source files and interfaces are automatically created and placed in ProjectBuilder. **Figure 1** shows ProjectBuilder with a new application. Double-clicking lunch.nib will open the interface file for the lunch application in InterfaceBuilder.

Figure 1.

InterfaceBuilder creates and edits the user interfaces for OPENSTEP applications. The interfaces are stored in "nib" files. Buttons, windows, lists, and other UI elements can be added to an application graphically. Some of these steps are hard to explain if you don't have OPENSTEP in front of you, but once you have OPENSTEP, you will see that this is all straight-forward pointing and clicking.

The lunch application will contain a text field, a button, and a scrolling LunchView object. The text field and button are already provided by InterfaceBuiler and can be dragged into the window from the palette. The LunchView object is a placed by dragging a CustomView and resizing it. InterfaceBuilder's menu item called Group In ScrollView moves the custom view into a scrollview. InterfaceBuilder has an Inspector window for setting attributes on objects. The inspector is used to fine tune the locations of the controls. The inspector also allows us to disable the Eat button. The resulting window is shown in **Figure 2**.

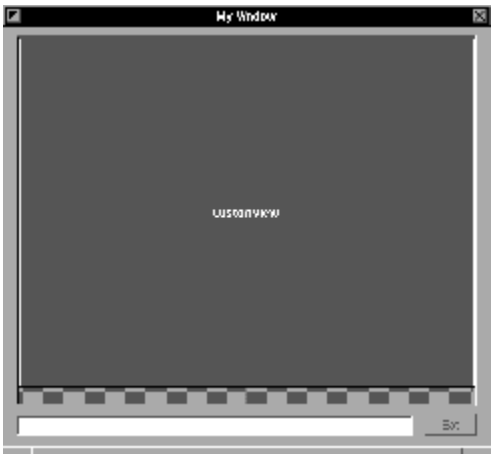

Figure 2.

User Interface Control

The lunch application uses two custom objects. InterfaceBuilder allows custom objects to be created and automatically integrated into ProjectBuilder. When a nib is opened, a window titled with the name of the nib appears in the lower left. This window lists all the non-graphical objects in the nib file, as well as the windows. Custom objects are created under the classes tag.

Figure 3.

Every nib file is owned by an object, which is typically the object created just before the nib is loaded to hook the nib into the application. This object is represented by the File's Owner icon in **Figure 3**, and can be set in the Inspector. For lunch.nib, the File's Owner is an Application object because this is the starting nib. A subclass of the Application could be made the File's Owner, but delgation works better.

Some objects, such as Window, Text, and Application, have hooks built into them to delegate functionality to other objects. We will create a custom object to act as our Application's delegate. Delegates modify the functionality of core objects by implementing methods. They need only implement some of the methods. If a method is not implemented, then the core object will know not to call it. Objective-C allows run-time checking of what routines are implemented, so delegates can be very light-weight objects. The LunchObject is the delegate to both the Application and a text field.

Custom objects are created by selecting Classes from the tags. The lower left window will now look like **Figure 4**. In the bottom of the window is a pull-down menu called Operations. This menu contains the operators used to create custom objects. The subclass operator will create a subclass of Object. The new object starts with the name MyObject, but should be changed to LunchObject for this project. LunchObject will implement methods for controlling the nib's custom interface objects. Outlets in the LunchObject are required for the window, edit button, lunch view and text field. Create 4 outlets and name them window, lunchField, lunchView, and lunchButton. Also create an action called eat. The unparse operator will create a template, and puts the source files in ProjectBuilder. Selecting the instantiate operator will create an instance of LunchObject.

Figure 4.

Holding down the control key, drag from the File's Owner field to the LunchObject. The inspector shows what connections are possible. Selecting the delegate outlet and clicking OK in the Inspector hooks these objects together. No code change is necessary to hook objects together. Dragging from the LunchObject to the window, text field, custom view, and button allows us to set the LunchObjects outlets. To set the buttons action, drag from the button to the LunchObject and set the targets action to be eat.

To enable and disable the Eat button depending on if there is text in the text field, make the LunchObject the delegate of the text field. (One object can be the delegate to many other objects.) Drag from the text field to the LunchObject to set the delegate. Another nice feature for users is to be able to press return in the text field and have the Eat button hit. This is done by dragging from the text field to the button and selecting target:performClick. Again, no code changes are necessary.

Whenever text is changed, we must enabled or disable the button. The delegate method that best handles this is -textDidGetKeys:isEmpty, implemented in Listing 1. This method is called whenever the text field receives keys. For performance reasons, it is not called each time a key is pressed, but for our purposes it will work very well.

```
Listing 1: User Interface code
- textDidGetKeys:sender isEmpty:(BOOL)flag
{
 [lunchButton setEnabled:!flag];
 return self;
}
```

The second custom object is our LunchView. LunchView is created very much like the LunchObject, but it is a subclass of View. It has no outlets or actions, but is unparsed like LunchObject. Instead of instantiating a copy, we click the CustomView we created earlier and use the inspector to set the custom view to be our LunchView.

Once the nib is saved, the user interface is done.

Networking

In Objective-C, run-time binding allows two objects with completely different inheritences to implement the same method. However, this flexibility has some costs. Run-time lookups take slightly longer, the compiler cannot catch bad calls, and the caller has to wait to see if the receiver is going to respond. Objective-C partially solves these problems by implementing protocols. A protocol is a list of methods that objects must implement. There is still a cost for run-time lookups, but the compiler will catch bad calls, and the caller can know if a value will be returned. With networking, not having to wait for a response can make a big difference. Our LunchObject supports two protocols, LunchClientMethods and LunchServerMethods.

Inter-Process communication is very easy under OPENSTEP. The first step is forming a connection between two processes. LunchObject is the delegate of Application and gets an appDidInit call when the application is done launching. Likewise, it gets an appWillTerminate call before the application quits. If a connection breaks, senderIsInvalid will be called. These three routines are in Listing 2.

```
Listing 2: Network connection
- appDidInit:sender
{
 remoteHub =
 [NXConnection connectToName:"Lunch" onHost:"*"];
 if (remoteHub)
 {
    //We will be a client
 ourServer  = [remoteHub connectionForProxy];
 [remoteHub
 setProtocolForProxy:@protocol(LunchServerMethods)];
 isServer = NO;
 }
 else
 {
    //We are first to launch, so we are the server
 ourServer =
 [NXConnection registerRoot: self withName:"Lunch"];
 remoteHub  = self;
 isServer = YES;
 clientList = [[List alloc] init];
 }

 [ourServer registerForInvalidationNotification:self];
 [ourServer runFromAppKit];
 [remoteHub addClient:self];
 [self updateEveryone];

 return self;
}

- appWillTerminate:sender
{
 [remoteHub removeClient:self];
 return self;
}

- senderIsInvalid:sender
{
 if ( sender != self )
 {
    //server died, we all have to go away
 remoteHub = nil;
 [NXApp terminate:self];
 }
else if ( isServer )
 [self removeClient:sender];

 if ( isServer )
```

```
 [self updateEveryone];

 return self;
}
```

LunchObject will act as both server and client. Each LunchObject checks the network for any other LunchObject, if it doesn't find one, then it will register itself as the server. By checking the host *, we check all machines on the network. If we knew a server was running on one particular machine, then we could specify that machine as the host.

Registering self as the root object will cause this object to be distributed over the network. The next LunchObject that runs will check the network, find the first LunchObject registered, and receive a proxy to the first LunchObject. A proxy is an object that pretends to be another object. Any calls to the proxy go through Mach messages to the real object. The real object can return any data it wants, including other objects. New objects can be passed by copy or proxy. By default, the first object will send a proxy to the second object.

Once the two sides are communicating, the client tells the server about itself by asking the server to addClient on itself. It is passing a proxy of itself to the server. If a client quits, then it will remove itself from the server. The connection is also set to understand the LunchServerMethods protocol to avoid round-trip calls when possible.

Once the two sides are communicating, they must update all the clients. Listing 3 has the code for adding and removing clients, as well as the code for updating all the clients. One of the drawbacks of Inter-Process Communication is dead-locking. If one method calls another method on the other side, and that method has a callback to the first side, the two sides could be blocked waiting for the other to complete. Using perform:afterDelay methods prevent this. The perform:afterDelay will wait until the next time through the event loop and perform that method. The original calls will then not dead-lock. OPENSTEP's distributed objects are not too sensitive to dead-locking, but it is a good habit to always protect against it.

```
Listing 3: Client-server communications
- updateEveryoneAfterDelay
{
 [clientList makeObjectsPerform:@selector(updateStats)
 with:nil];
 return self;
}

- (void)updateEveryone
{
 [self perform:@selector(updateEveryone)
 with:nil afterDelay:1 cancelPrevious:YES];
 return;
}

- (void)addClient:client;
```

```
{
 [clientList addObjectIfAbsent:client];
 [self updateEveryone];
}

- (void)removeClient:client;
{
 [clientList removeObject:client];
 [self updateEveryone];
}
- (List *)clientList
{
 return clientList;
}
- (void)updateStats;
{
 [self perform:@selector(updateStatsAfterDelay) with:nil afterDelay:1
cancelPrevious:YES];
}
```

Whenever a client is added or removed, the server calls its updateEveryone method. That method will wait until the next event loop, call updateEveryoneAfterDelay. updateEveryoneAfterDelay calls updateStats in every client, which will call each clients updateStatsAfterDelay after a delay. updateStatsAfterDelay will rebuild everything. Whenever any client changes, it can call updateEveryone in the server and every client will update.

The last three routines needed for networking are to send the actual data. The eat: method saves the user's choice and updates all the clients. Each client's updateStatsAfterDelay method must find out where each user wants to eat by calling wantToEatWhere method. In our code, the updateStatsNow only passes the list of clients to the LunchView, the LunchView will call wantToEatWhere. These 3 routines are listed in Listing 4.

```
Listing 4
- eat:sender
{
 [where release];
 where = [NSString stringWithCString:[(TextField *)lunchField
stringValue]];
 [remoteHub updateEveryone];
 return self;
}
- updateStatsAfterDelay
{
 List *serverClientList;
 serverClientList = [remoteHub clientList];
 [lunchView updateLunchList:serverClientList];
 return self;
}

- (NSString *)wantToEatWhere;
{
 return where;
}
```

The choice of where to eat is stored in an NSString. NSString is part of the Foundation Kit instead of the AppKit. Intermingling different kits is usually transparant. Foundation Kit objects provide garbage collection. Any object can be set to be auto-released and the next iteration of the event loop will free all the objects in the pool. NSString returns a char * that will automatically be freed later.

LunchView

Displaying the lunch results is done in the LunchView. LunchView is a subclass of view and as such, knows about setting up the display system. The actual code to display is simple, but turning the distributed list into something easy to display is a bit more complicated. In C++, there would probably be 3 or 4 objects to do this, but in our example we will have only one helper object, LunchChoice. Objective-C objects tend to be larger than C++ objects. All functionality for viewing should be placed in the view object.

LunchChoice, our helper object, will store one restaurant choice and the number of times this restaurant has been requested. The code is in Listings 5 and 6. The LunchChoice stores the restaurant in an NSString, which will take care of garbage collecting for us. The + sign before some of the methods are the same as C++ static methods, but are called factory methods. LunchChoice uses these factory methods to store the MaxValue so LunchView can scale appropriately.

NEXTSTEP has a convention of using init... for initialaztion methods and free for freeing methods. OPENSTEP has a convention for releasing objects instead of freeing them. Releasing, rather than freeing, an object is used by the garbage collection system. The initLocation method will call [super init], then an NSString object. The free method is called whenever we are done with this object, and it releases the NSString allocatted as well as this object. The rest of the methods are straight-forward.

```
Listing 5: Lunch Choice Header
#import <appkit/appkit.h>
#import <foundation/foundation.h>

Object
{
 NSString *where;
 int    value;
}

+ ResetMaxValue;
+ (int)MaxValue;
- initLocation:(const char *)location;
- incrementValue;
- (NSString *)location;
- (int)value;
- free;
```

```
 LunchChoice Source
#import "LunchChoice.h"

(const char *)location
{
 [super init];
 where = [NSString stringWithCString:location];
 value = 0;
 return self;
}

- incrementValue;
{
 value++;
 if ( value > MaxValue )
 MaxValue = value;
 return self;
}

- (NSString *)location;
{
 return where;
}

- (int)value;
{
 return value;
}

- free;
{
 [where release];
 return [super free];
}
```

LunchView, being a subclass of view, has a standard initialization method called initFrame. initFrame will call [super init] and create storage for the list of choices. The free method will free all memory associated with LunchView. Unlike C++ with constructors and destructors, init and free methods must be manually called. These routines are in Listing 7.

Listing 7: LunchView initialization and free methods

```
- initFrame:(const NXRect *)frm;
{
 [super initFrame:frm];
 choices = [[List alloc] init];
 return self;
}
- free
{
 [choices freeObjects];
 [choices free];
 return [super free];
}
```

LunchView's updateLunchList method gets called whenever there is a change in restaurant. The list passed is located in the server process, but by this point, it is transparant to LunchView. Any call to the list passed in will act like a call to a local object, but be a bit slower. This routine deletes the existing list and recreates a new list from scratch. Not the most efficient algorithms, but works for our purposes. The updateLunchList method calls the getChoice method with a string value. The getChoice method will return a LunchChoice object of that name, creating one if necessary. Once a LunchChoice is found, it's value is incremented. When the new list has been parsed, the view is resized and redrawn. The list handling routines are in Listing 8.

Listing 8: LunchView's choice handling methods

```
- (LunchChoice *)getChoice:(const char *)location
{
 int    count;
 const  char*thisLocation;
 LunchChoice*thisChoice;

 for ( count = [choices count] - 1; count >= 0; count- )
 {
 thisChoice = [choices objectAt:count];
 thisLocation  = [[thisChoice location] cString];
 if ( !strcmp( thisLocation, location ) )
 break;
 }
 if ( count < 0 )
 {
 thisChoice =
 [[LunchChoice alloc] initLocation:location];
 [choices addObject:thisChoice];
 }
 return thisChoice;
}

- updateLunchList:lunchList;
{
 int    count;
 NSString *thisLocation;
 LunchObject*thisHandler;
 LunchChoice*thisChoice;

 [choices freeObjects];
 [LunchChoice ResetMaxValue];
 for (count = [lunchList count] - 1; count >= 0; count-)
 {
 thisHandler   = [lunchList objectAt:count];
 thisLocation  = [thisHandler wantToEatWhere];
 thisChoice = [self getChoice:[thisLocation cString]];
 [thisChoice incrementValue];
 }

 [self updateFrameSize:[choices count]];
 [self display];
```

```
 return self;
}
```

Frame resizing is done in the updateFrameSize method. When a view in a scrolling view is resized, everything works correctly. No extra code is needed. However, code to make sure the view doesn't shrink smaller than the visible area is necessary. Otherwise, there will be areas of the ScrollView that are not cleared correctly. This routine is in Listing 9.

```
Listing 9: Frame handling
- updateFrameSize:(int)numberOfColumns
{
 NXRect theFrame, visibleBounds;

 [self getFrame:&theFrame];
 [superview getVisibleRect:&visibleBounds];

 if (visibleBounds.size.width
 < numberOfColumns * kWidthPerColumn)
 theFrame.size.width = numberOfColumns * kWidthPerColumn;
 else
 theFrame.size.width = visibleBounds.size.width;
 [self setFrame:&theFrame];
 return self;
}
```

The drawSelf method is responsible for all drawing. In general, the rectangle to be updated is passed as a pointer to this method and a rectCount of 1. However, for optimization, it is possible to pass an additional 2 rects to specify a non-rectangular area to update with a rectCount of 3. The first rectangle, however, must always contain a rect that can update the entire window.

When drawSelf is called the postscript context has been set for the drawing. Postscript commands sent to the postscript server will be processed and displayed. There is a buffering built in, so there will be little to no flicker. If there is any flicker, disabling the flushing is a single call. Turning the flushing back on and flushing the window will blast the pixels in one shot. LunchView did not need this extra bit of code.

The first step is to set the font and color LunchView is going to use. The font chosen is a screenFont, but if we want to print this window, OPENSTEP will automatically switch to a printer font. Next, a rectangle to draw each choice is created. Finally, the choices are iterated and information is displayed. The NXRectFill shows a weighted value for each choice and the Psshow displays the choice. Listing 10 has the source to drawSelf and Figure 3 has the final screenshot.

```
Listing 10:
- drawSelf:(const NXRect *)rects :(int)rectCount;
{
 int    count;
 NXRect thisRect;
```

```
 PSsetgray( NX_BLACK );
 [[[Font newFont:"Helvetica"
 size:10.0 matrix:NX_IDENTITYMATRIX] screenFont] set];
 thisRect.origin.y = bounds.origin.y + kOffset*2;
 thisRect.origin.x = bounds.origin.x + kOffset;
 thisRect.size.width = thisRect.size.width + kWidthPerColumn -
kOffset*2;

 NXEraseRect( rects );
 for ( count = [choices count] - 1; count >= 0; count- )
 {
 LunchChoice*thisChoice = [choices objectAt:count];

 PSmoveto(  thisRect.origin.x + kOffset,
 bounds.origin.y + kOffset );
 PSshow( [[thisChoice location] cString] );
 thisRect.size.height =
 (bounds.size.height - kOffset*3)*
 ((float)[thisChoice value])/
 ((float)[LunchChoice MaxValue]);
 NXRectFill( &thisRect );
 thisRect.origin.x += kWidthPerColumn;
 }

 return self;
}
```

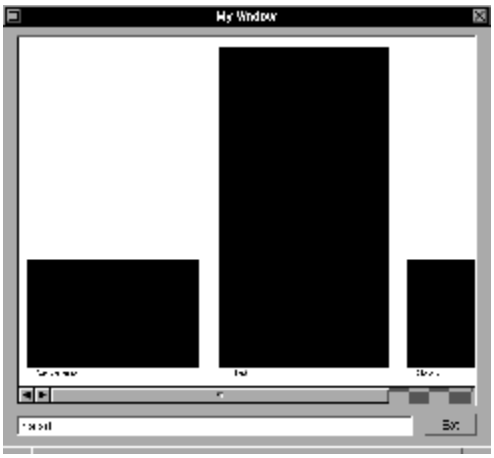

Figure 5.

What else can I do that is easy?

From Interface Builder, printing and spell checking can be added with no coding at all. Setting a font would require an additional method, but roughly only 5 lines of code would be added or changed.

Setting up a chat line so people can talk while this is going on wouldn't be difficult at all. The list of LunchObjects is already distributed, so a list of chat objects can be easily added.

Perhaps statistics should be kept. OPENSTEP provides an NXStringTable object that allows storage to a file of a key value pair. The NXStringTable object makes simple hashing a dream.

When marketing droids say that developing in OPENSTEP is 3 times as fast as conventional development, everyone doubts them. From personal experience, I've found it to be an almost 10-fold increase. This entire application, including complete networking code, took me less than 4 hours to write. How long would it take to write this same application on a Macintosh or under Windows?